Towards Efficient and Expressive Runtime Monitors

Understanding Complex Systems by Monitoring their Execution

Klaus Havelund
Jet Propulsion Laboratory
California Institute of Technology

April 20, 2012

In Collaboration With

- University of Manchester, UK
  - Howard Barringer (Professor)
  - Giles Reger (Ph.D. student)
  - David Rydeheard (Dr.)
- University of Grenoble, France
  - Yliès Falcone (Associate Professor)

Contents

Runtime Verification

- *Monitoring* the runtime behavior of a system with respect to a user-defined *property*
- Need to *instrument* system to select relevant events
- *Online* or *offline* (log-files)
- If online
  - verdict returned after each event
  - can give feedback to steer the system

Runtime Verification in Theory

- *Events* record runtime behavior
  - snapshots of state or actions performed
- A finite sequence of events is a *trace* τ
- A *property* ϕ denotes an *event language* $\mathcal{L}(\phi)$ (a set of traces)
- τ satisfies ϕ iff $\tau \in \mathcal{L}(\phi)$

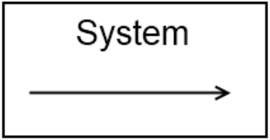
Runtime Verification in Theory

- Should detect success/failure before end of trace
- Standard approach is to use *four-valued verdict domain*
- Consider all possible extensions of a trace

	current trace τ	all suffixes σ	Action
1	$\tau \in \mathcal{L}(\varphi)$	$\tau\sigma \in \mathcal{L}(\varphi)$	stop with Success T
2	$\tau \in \mathcal{L}(\varphi)$	unknown	carry on monitoring T_p
3	$\tau \notin \mathcal{L}(\varphi)$	$\tau\sigma \notin \mathcal{L}(\varphi)$	stop with Failure F
4	$\tau \notin \mathcal{L}(\varphi)$	unknown	carry on monitoring F_p

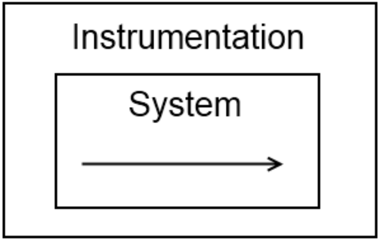
Runtime Verification in Practice

- Start with a system to monitor



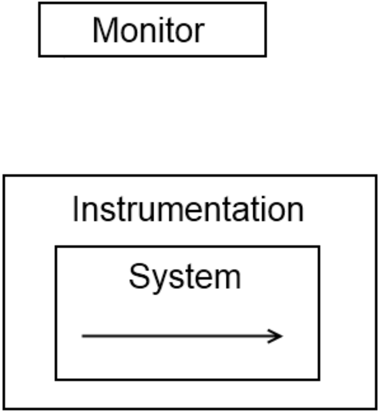
Runtime Verification in Practice

- *Instrument* the system to record relevant events



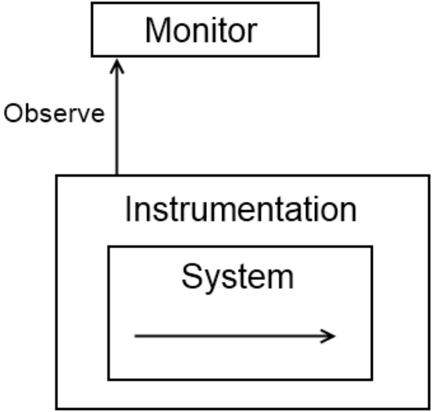
Runtime Verification in Practice

- *Generate* a monitor from the property



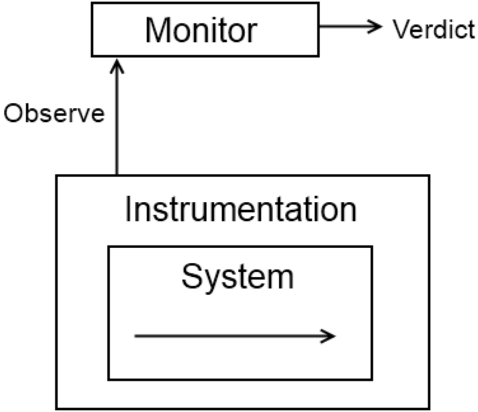
Runtime Verification in Practice

- *Dispatch* each received event to the monitor.



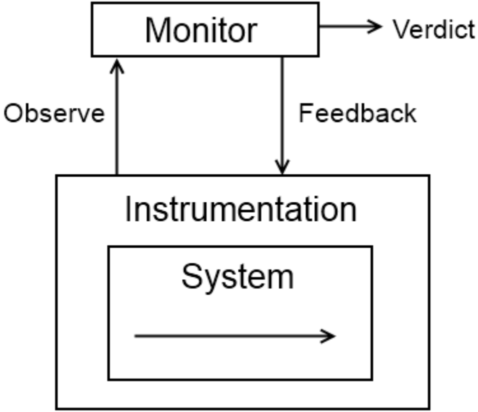
Runtime Verification in Practice

- Compute a *verdict* for the trace received so far.



Runtime Verification in Practice

- Possibly generate *feedback* to the system.



Runtime Verification Applications

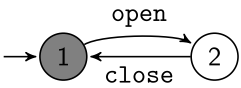
- Detect erroneous behavior after deployment (fault protection)
- Detect intrusion after deployment (security)
- Monitor as part of testing before deployment (test oracles)
- Program understanding

Some Context

- Field started with *propositional* monitoring
  - events are just strings
- Recently moved to *parametric* monitoring
  - events include data values
- Solutions exist spanning the two classical dimensions
  - *Expressiveness* of specification language
  - *Efficiency* of monitoring algorithm
- This work is looking for the right combination

The Propositional Approach : An Example

- Record *propositional* events, for example
  - `open`, `close`
- Define a property over propositional events, for example
  - LTL (finite-trace) $\Box(\texttt{open} \rightarrow \bigcirc(\neg\texttt{open}\ \mathcal{U}\ \texttt{close}))$
  - RE $(\texttt{open}.\texttt{close})^*$
  - DFA



- Check if each trace prefix is in the language of the property

Going Parametric

- Consider the code

```
File f1 = new File("manual.pdf");
File f2 = new File("readme.txt");
f1.open();
f2.open();
f2.close();
f1.close();
```

Going Parametric

- Consider the code

```
File f1 = new File("manual.pdf");
File f2 = new File("readme.txt");
f1.open();
f2.open();
f2.close();
f1.close();
```

- Say we just focus on propositional events

open.open.close.close

Going Parametric

- Consider the code

```
File f1 = new File("manual.pdf");
File f2 = new File("readme.txt");
f1.open();
f2.open();
f2.close();
f1.close();
```

- Say we just focus on propositional events

`open.open.close.close`

- No good, we want to *parameterize* events with data values and use those values in the specification

Going Parametric

- Consider the code

```
File f1 = new File("manual.pdf");
File f2 = new File("readme.txt");
f1.open();
f2.open();
f2.close();
f1.close();
```

- Say we just focus on propositional events

`open.open.close.close`

- No good, we want to *parameterize* events with data values and use those values in the specification
- Instead record the parametric trace

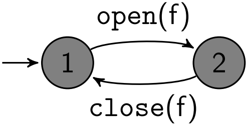
open(manual.pdf).open(readme.txt).close(readme.txt).close(manual.pdf)

Parametric Properties

- Using the events
  - `open`(f) when file f is opened
  - `close`(f) when file f is closed

Parametric Properties

- Using the events
  - `open`(f) when file f is opened
  - `close`(f) when file f is closed
- the property becomes

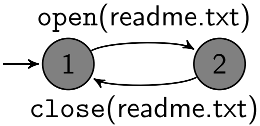


Instantiating Parametric Property

- Let f = readme.txt (a binding)

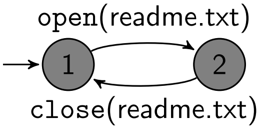
Instantiating Parametric Property

- Let f = readme.txt (a binding)
- Instantiated property becomes



Instantiating Parametric Property

- Let f = readme.txt (a binding)
- Instantiated property becomes

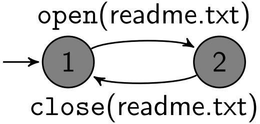


- Given parametric trace

open(manual.pdf).open(readme.txt).close(readme.txt).close(manual.pdf)

Instantiating Parametric Property

- Let f = readme.txt (a binding)
- Instantiated property becomes



- Given parametric trace

open(manual.pdf).open(readme.txt).close(readme.txt).close(manual.pdf)

- project to

open(readme.txt).close(readme.txt)

From Parametric to Quantified

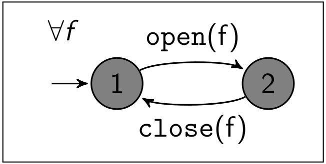
- Where do bindings come from?

From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

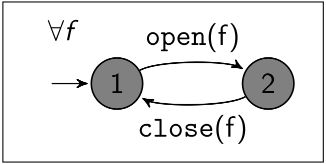
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property



From Parametric to Quantified

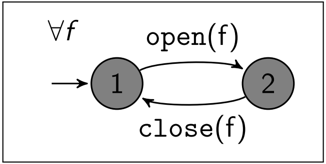
- Where do bindings come from?
- quantify over variables in parametric property



- Universal and existential quantification

From Parametric to Quantified

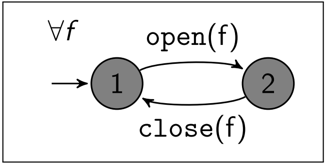
- Where do bindings come from?
- quantify over variables in parametric property



- Universal and existential quantification
- What is the domain of quantification? (choice)

From Parametric to Quantified

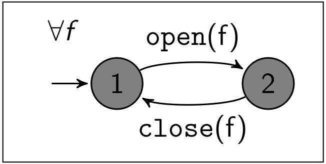
- Where do bindings come from?
- quantify over variables in parametric property



- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace

From Parametric to Quantified

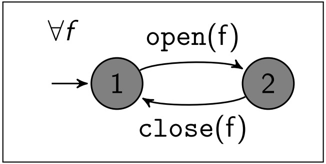
- Where do bindings come from?
- quantify over variables in parametric property



- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
- How? (choice)

From Parametric to Quantified

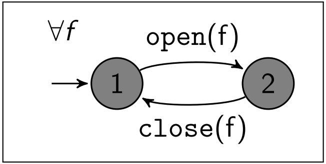
- Where do bindings come from?
- quantify over variables in parametric property



- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
- How? (choice)
- *Match* events in parametric property with events in trace

From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property



- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
- How? (choice)
- *Match* events in parametric property with events in trace
- open(f) matches open(readme.txt) and open(manual.pdf)

  $[f \mapsto \{$ readme.txt, manual.pdf $\}]$

Using Data Values : A Task Monitoring Example

- All tasks must end phases in increasing order

Using Data Values : A Task Monitoring Example

- All tasks must end phases in increasing order
- Events monitored: `end`(*task*,*phase*)

Using Data Values : A Task Monitoring Example

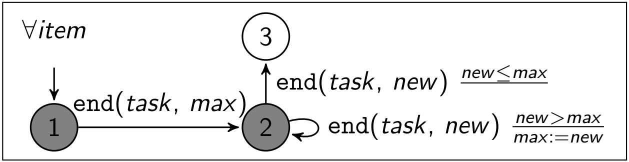
- All tasks must end phases in increasing order
- Events monitored: `end`(*task*,*phase*)
- Example trace

  `end`(42, 5).`end`(42, 6).`end`(42, 3)

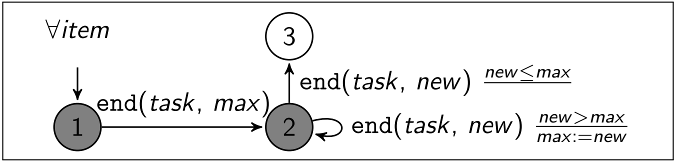
Using Data Values : A Task Monitoring Example

- All tasks must end phases in increasing order
- Events monitored: end(*task*,*phase*)
- Example trace

  end(42, 5).end(42, 6).end(42, 3)

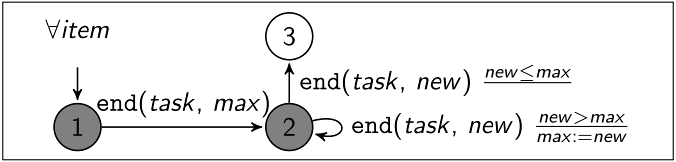


Using Data Values : A Task Monitoring Example



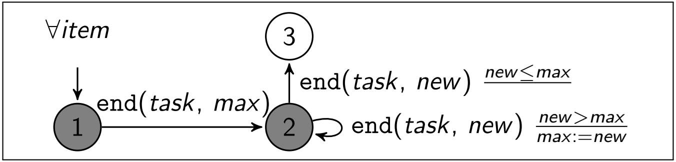
- trace: `end(42, 5).bid(42, 6).bid(42, 3)`

Using Data Values : A Task Monitoring Example



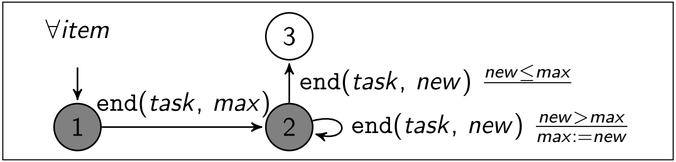
- trace: `end`(42, 5).`bid`(42, 6).`bid`(42, 3)
- domain is $[task \mapsto \{\ 42\ \}]$

Using Data Values : A Task Monitoring Example

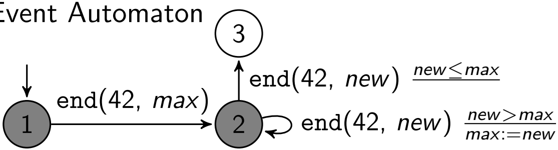


- trace: end(42, 5).bid(42, 6).bid(42, 3)
- domain is $[task \mapsto \{\ 42\ \}]$
- partially instantiate parametric property with $[task \mapsto 42]$

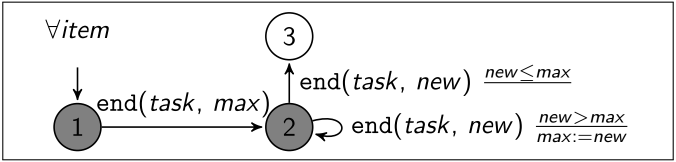
Using Data Values : A Task Monitoring Example



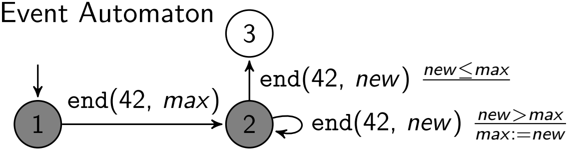
- trace: `end(42, 5).bid(42, 6).bid(42, 3)`
- domain is $[task \mapsto \{\ 42\ \}]$
- partially instantiate parametric property with $[task \mapsto 42]$
- to get Event Automaton



Using Data Values : A Task Monitoring Example

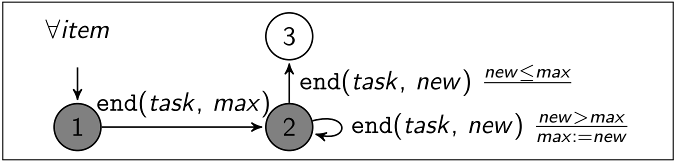


- trace: `end(42, 5).bid(42, 6).bid(42, 3)`
- domain is $[task \mapsto \{ 42 \}]$
- partially instantiate parametric property with $[task \mapsto 42]$
- to get Event Automaton

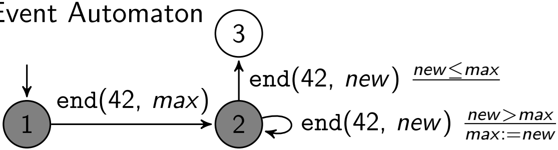


- keep local state per instantiated parametric property

Using Data Values : A Task Monitoring Example



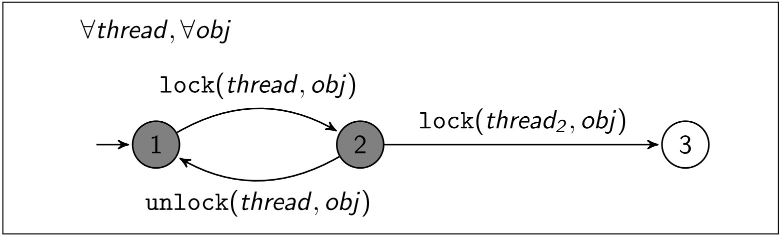
- trace: end(42, 5).bid(42, 6).bid(42, 3)
- domain is [$task \mapsto \{ 42 \}$]
- partially instantiate parametric property with [$task \mapsto 42$]
- to get Event Automaton



- keep local state per instantiated parametric property
- treat quantified and unquantified variables differently

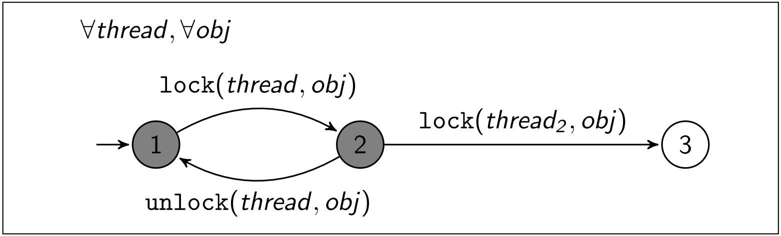
Another Example

- Every object can only be locked once at any one time



Another Example

- Every object can only be locked once at any one time



- `lock` is used with two different lists of formal parameters

Event Automata : Definition

$$
\begin{array}{lcl}
Bind & = & [Var \rightharpoonup Val] \\
Guard & = & [Bind \rightarrow \mathbb{B}] \\
Assign & = & [Bind \rightarrow Bind]
\end{array}
$$

Definition (Event Automaton)

An Event Automaton $\langle Q, \mathcal{A}, \delta, q_0, F \rangle$ is a tuple where

- Q is the set of states,
- $\mathcal{A} \subseteq Event$ is the alphabet,
- $\delta \in (Q \times \mathcal{A} \times Guard \times Assign \times Q)$ is the transition set,
- q_0 is the initial state, and
- $F \subseteq Q$ is the set of final states.

Quantified Event Automata : Definition

Definition (Quantified Event Automaton)

A QEA is a pair $\langle \Lambda, \mathrm{E} \rangle$ where

- E is an EA, and
- $\Lambda \in (\{\forall, \exists\} \times \texttt{variables}(\mathrm{E}) \times \mathit{Guard})^*$ is a list of quantified variables with guards.

Monitoring QEA

- Previous description is big-step (whole trace)

Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time

Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction

Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
  - Build up domain on the fly
  - Generate new bindings on the fly
  - Track configurations associated with bindings
  - Check acceptance

Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
    - Build up domain on the fly
    - Generate new bindings on the fly
    - Track configurations associated with bindings
    - Check acceptance
- Efficient algorithms

Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
  - Build up domain on the fly
  - Generate new bindings on the fly
  - Track configurations associated with bindings
  - Check acceptance
- Efficient algorithms
  - Lookup relevant monitors from event
  - Data-structures to deal with *matching*

TraceContract

- An internal Scala DSL (API)

TraceContract

- An internal Scala DSL (API)
- Expressive and easy to implement and modify

TraceContract

- An internal Scala DSL (API)
- Expressive and easy to implement and modify
- Based on formula rewriting: $p\,\mathcal{U}\,q = q \vee (p \wedge \bigcirc(p\,\mathcal{U}\,q))$

TraceContract

- An internal Scala DSL (API)
- Expressive and easy to implement and modify
- Based on formula rewriting: $p\,\mathcal{U}\,q = q \vee (p \wedge \bigcirc(p\,\mathcal{U}\,q))$
- (To be) used by LADEE:
  Lunar Atmosphere and Dust Environment Explorer

The Task Monitor Example

```
trait Event
case class End(task: Int, step: Int) extends Event

class TaskMonitor extends Monitor[Event] {
  always {
    case End(task, step1) =>
      watch {
        case End(`task`, step2) => step2 > step1
      }
  }
}
```

Analyzing a Trace

```
object Test extends Application {
  val m = new TaskMonitor

  val trace = List(
    End(1, 2),
    End(2, 1),
    End(1, 3),
    End(2, 2),
    End(1, 1))

  m.verify(trace)
}
```

Implementation of TraceContract

```
trait Monitor[E] extends RuleSystem {
  var current: state = True

   trait state {
     def apply(e: E): state
     def and(that: state) = And(this, that) reduce
     def or(that: state) = Or(this, that) reduce
  }

  type Body = PartialFunction[E, state]

  case class watch(b: Body) extends state {
     def apply(e: E) = if (b.isDefinedAt(e)) b(e) else this
  }
   ...
}
```

Implementation using Rewriting

```
case class repeat(b: Body) extends state {
  def apply(e: E) =
    if (b.isDefinedAt(e)) And(b(e), this) reduce else this
}

def init(s: state) {current = s}

def always(b: Body) = init(repeat(b))

def apply(e: E) {
  current = current(e)
  if (current == False) println("*** safety violation")
}
```

ScalaRules

- An internal Scala DSL (API)

ScalaRules

- An internal Scala DSL (API)
- Implements the RETE algorithm for rule-based systems

ScalaRules

- An internal Scala DSL (API)
- Implements the RETE algorithm for rule-based systems
- Efficient pattern matching algorithm for production rule systems

ScalaRules

- An internal Scala DSL (API)
- Implements the RETE algorithm for rule-based systems
- Efficient pattern matching algorithm for production rule systems
- Purpose is to investigate relevance for runtime verification

The Lock Monitor Example

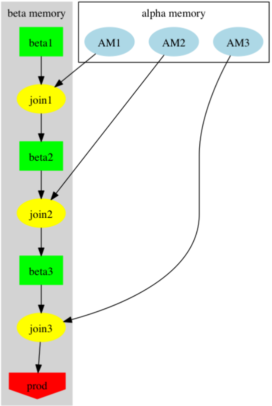
```
class LockMonitor extends ScalaRules {
  rule("goodLock") when
    exists ('kind->"lock", 'thread->'t, 'obj->'o) then
   add('kind->"Locked", 'thread->'t, 'obj->'o)

  rule("badLock") when
    exists ('kind->"lock", 'thread->'t1, 'obj->'o) and
    exists ('kind->"Locked", 'thread->'t2, 'obj->'o) then
    error

  rule("unlock") when
    exists ('kind->"unlock", 'thread->'t, 'obj->'o) and
    exists ('x)('kind->"Locked", 'thread->'t, 'obj->'o) then
   rem('x)
}
```

Running It

```
object Test extends Application {
  val r = new LockMonitor
  r.addFact('kind->"lock", 'thread->1, 'obj->42)
  r.addFact('kind -> "unlock", 'thread->1, 'obj->42)
  r.addFact('kind -> "lock", 'thread->1, 'obj->42)
  r.addFact('kind -> "lock", 'thread->2, 'obj->42)
}
```

RETE Network for a Rule

rule: $a(x), b(x, y), c(x, y) \Rightarrow action$



Future Work

- Theoretic foundations
  - QEA
  - TraceContract
- Implementation
  - Implement efficient monitoring algorithm for QEA
  - Explore utility of RETE algorithm and modifications
- Application
  - Support application of TraceContract within LADEE mission
  - Apply to logs for JPL mission
- Related topics
  - Inferring properties from runs
  - Annotating logs (Rajeev Joshi)
  - Program visualization

Conclusion

- Efficient state of the art systems lack expressiveness
- We attempt to increase expressiveness while staying efficient
- Result should be efficient and expressive RV system
- RV is a scalable way to understand complex systems
- Scala as prototyping language + internal DSLs speed up development

Publications

- **Aspect-Oriented Instrumentation with GCC**
  J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok
  RV 2010, St. Julians, Malta.
- **TraceContract: A Scala DSL for Trace Analysis**
  H. Barringer and K. Havelund
  FM 2011, Limerick, Ireland
- **Runtime Verification with State Estimation**
  S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok
  RV 2011, San Francisco, California, USA.
- **Quantified Event Automata:**
  Towards Expressive and Efficient Runtime Monitors
  H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard
  Submitted for publication, under review. March 2012.